Ordered Fast Fourier Transforms on a Massively Parallel Hypercube Multiprocessor

by

Charles Tong [1,3] and Paul N. Swarztrauber [2,3]

J. Parallel and Dist. Comput., **12**(1991), pp. 50-59

ABSTRACT

We examine design alternatives for ordered FFT algorithms on massively parallel hypercube multiprocessors such as the Connection Machine. Particular emphasis is placed on reducing communication which is known to dominate the overall computing time. To this end we combine the order and computational phases of the FFT and also use sequence to processor maps that reduce communication. The class of ordered transforms is expanded to include any FFT in which the order of the transform is the same as that of the input sequence. Two such orderings are examined, namely, "standard-order" and "A-order" which can be implemented with equal ease on the Connection Machine where orderings are determined by geometries and priorities. If the sequence has $N = 2^r$ elements and the hypercube has $P = 2^d$ processors then a standard-order FFT can be implemented with $d+r/2+1$ parallel transmissions. An A-order sequence can be transformed with $2d-r/2$ parallel transmissions which is $r-d+1$ fewer than the standard order. A parallel method for computing the trigonometric coefficients is presented that does not use trigonometric functions or interprocessor communication. A performance of 0.9 GFLOPS was obtained for an A-order transform on the Connection Machine.

1 Department of Computer Science, University of California at Los Angles, Los Angles, California 90024-1596.

2 National Center for Atmospheric Research, Boulder, Colorado 80307, which is sponsored by the National Science Foundation.

3 This work was supported by the NAS Systems Division via Cooperative Agreement NCC 2-387 between NASA and the University Space Research Association (USRA). It was performed while the authors were visiting the Research Institute for Advanced Computer Science (RIACS), NASA Ames Research Center, Moffett Field, CA 94035.

1. Introduction

The increased availability of various parallel architectures poses many challenges for algorithm development. One notable example is the Fast Fourier Transform (FFT) with many variants that are targeted for different types of computers. The main difference between these variants is the order of the intermediate sequences which have been selected to favor certain architectural characteristics. For example, orderings that result in long vectors with unit stride are selected for vector computers [6]. Orderings that minimize communication are selected for hypercube multiprocessors [7]. Interprocessor communication is the major source of performance degradation on hypercube multiprocessors.

In this paper we examine efficient implementation of ordered FFTs on massively parallel hypercube computers such as the Connection Machine. The concept of an ordered transform is expanded to include any transform in which the ordering of the input sequence matches that of its transform. This is a reasonable consideration on the Connection Machine where orderings can be selected with equal ease by the specification of geometries and priorities. Two "ordered" transforms are considered, namely, standard-order and A-order transforms. These transforms differ in communication complexity and their suitability will likely depend on the application. If a standard-order transform is not required then an A-order transform with less communication may be appropriate.

The standard-order transform was considered earlier [7] where it was demonstrated that a sequence with $N = 2^r$ elements could be transformed with $r/2+d+1$ parallel transmissions on a hypercube with $P = 2^d$ processors if $d > r/2$. Here we show that an A-order transform can be computed with $2d-r/2$ parallel transmissions. Both orderings belong to the class of orderings called index-digit permutations [1]. Besides reducing the amount of communication, we also show that this algorithm facilitates the parallel computation of the trigonometric coefficients without evaluating the trigonometric functions or interprocessor communication. Although we will consider only "ordered" transforms in the expanded sense, it is important to note that an unordered transform can be computed with only d parallel transmissions.

In section 2, we begin with a class of orderings called index-digit permutations. In particular, we review the concept of i-cycle which is central to the implementation of ordered hypercube FFT as well as the general index-digit permutation. In particular, we examine the standard-order FFT and the A-order transform which is yet to be defined. In section 3, we first discuss different ways of computing the trigonometric coefficients and then present a new parallel method for the direct computation of the

trigonometric coefficients. Next we show that this method is particularly suited to a hypercube implementation using i-cycles. The performance results of these FFTs are presented in section 4.

2. Parallel Hypercube FFTs

2.1 Introduction

In this paper we consider the implementation of ordered FFTs on hypercube multiprocessors. It is assumed that the number of physical processors is $P = 2^d$ where d is the dimension of the hypercube. Each processor has its own local memory (also called distributed-memory system). It is also assumed that the number of elements to be transformed is $N = 2^r$ and that N/P is a small constant (massively parallel version of the original hypercube FFT [7]). Moreover, if $N/P > 2$ (number of elements is more than twice the number of physical processors), the elements are mapped to virtual processors which then contain exactly two elements, (after the Connection Machine model). It is known that interprocessor communication consumes a substantial amount of time and hence its minimization is of primary concern. Communication between virtual processors located in the same physical processor does not contribute to interprocessor communication. Throughout the text we will use the following notation.

If x_n has $N = 2^r$ elements then it can be mapped into the multidimensional array $x(i_{r-1}, \cdots, i_0)$ where $i_{r-1}i_{r-2}\cdots i_0$ is the binary form of n. The FFT can then be loosely described as a sequence of 2^{r-1} transforms of length two in each of r dimensions. An example for the case $N = 16$ is given in Table I. below.

Table I : Intermediate Orderings for Cooley-Tukey FFT,
N=16, using Subscript Notation

$x(i_0,i_1,i_2,i_3)$
$X^{(1)}(i_0,i_1,i_2,k_3)$
$X^{(2)}(i_0,i_1,k_2,k_3)$
$X^{(3)}(i_0,k_1,k_2,k_3)$
$X^{(4)}(k_0,k_1,k_2,k_3)$
$X^{(4a)}(k_3,k_2,k_1,k_0)$

The original sequence is given as the first entry in Table I. The transform in the dimension i_3 is designated by replacing i_3 by k_3 in the second entry. Subsequent

multiple 1-D transforms correspond to subsequent entries in Table I. The FFT requires the multiple 1-D transforms to be computed in the order of decreasing indices, i.e., i_3, i_2, i_1, and i_0. The last entry corresponds to the bit-reversal that is necessary to order the FFT. Between each of the multiple 1-D transforms the sequence x_n is multiplied by certain roots of unity. For example, $X^{(1)}(i_0,i_1,i_2,k_3)$ is computed from

$$X^{(1)}(i_0,i_1,i_2,0) = x(i_0,i_1,i_2,0) + x(i_0,i_1,i_2,1) \tag{1}$$

$$X^{(1)}(i_0,i_1,i_2,1) = \omega^{i_0 i_1 i_2}[x(i_0,i_1,i_2,0) - x(i_0,i_1,i_2,1)] \tag{2}$$

where $\omega = e^{-i\pi/4}$.

We will adopt the binary notation in place of the subscript notation to avoid conversions between the two. Table II is the binary equivalent of the subscript notation that is used in Table I. Element locations are then given directly in binary form.

Table II : Intermediate Orderings for Cooley-Tukey FFT,
N=16 Binary Notation

$x(\, i_3\, i_2\, i_1\, i_0\,)$
$X^{(1)}(\, k_3\, i_2\, i_1\, i_0\,)$
$X^{(2)}(\, k_3\, k_2\, i_1\, i_0\,)$
$X^{(3)}(\, k_3\, k_2\, k_1\, i_0\,)$
$X^{(4)}(\, k_3\, k_2\, k_1\, k_0\,)$
$X^{(4a)}(\, k_0\, k_1\, k_2\, k_3\,)$

The last two entries in Table II correspond to a reordering in which the element in position $k_3k_2k_1k_0$ binary is moved to position $k_0k_1k_2k_3$. This illustrates the advantage of the binary notation which provides the locations directly without reversing the order of the subscripts.

The last entry in Tables I and II is an example of an index-digit permutation [1], called a bit-reversal. Other examples include the perfect shuffle and matrix transpositions. The time required for communication is known to contribute substantially to the overall computing time. It is also known to depend significantly on how the sequence x_n is mapped to the processors. We will begin with perhaps the most common mapping in which the first N/P elements are mapped to the first processor, the second N/P elements are mapped to the second processor and so forth.

Definition 1 : A **standard** sequence to processor map $x(\, i_{r-1} \cdots i_{r-d} \mid i_{r-d-1} \cdots i_0\,)$ is one in which the element x_n with $n = i_{r-1}i_{r-2}\cdots i_0$ (binary) has **address** $i_{r-d-1}i_{r-d-2}\cdots i_0$ in **processor number** $i_{r-1}i_{r-2}\cdots i_{r-d}$.

Both a processor number and address are required to identify a particular element in the sequence. The partition $\mid$ is introduced for expository purposes to separate the address on the right from the processor number on the left. For example if $r = 4$ and $d = 2$ then the element x_n with $n = i_3i_2i_1i_0$ has address i_1i_0 and is located in processor number i_3i_2 and the mapping is designated by $x(i_3\ i_2 \mid i_1\ i_0)$.

Definition 2 : An **index-digit permuted** sequence to processor map is one in which the indices i_j are permuted. That is, the element x_n with $n = i_{r-1}i_{r-2}\cdots i_0$ (binary) has address $i_{m(r-d-1)}i_{m(r-d-2)}\cdots i_{m(0)}$ in processor number $i_{m(r-1)}i_{(r-2)}\cdots i_{(r-d)}$ where $m(j)$ is an arbitrary permutation of the integers $0, \cdots, r-1$.

From the last two entries in Table II it is evident that a method will be needed for converting between index-digit permuted maps on the hypercube. To that end we introduce a specific class of communication tasks.

Definition 3 : An **i-cycle** is an index-digit permutation of x_n in which the most significant digit of the address (called the **pivot**) is exchanged with any other digit, either in the address or the processor number.

For example, if a standard sequence to processor map is used for x_n, an i-cycle is a reordering that exchanges the digit in position $r-d-1$ with any other digit. Two i-cycle examples are given in Table III.

Table III : Sample i-cycles for the case $d = 2$ and $r = 4$

$X(\, i_3\ i_2 \mid i_1\ i_0\,)$
$X(\, i_3\ i_2 \mid i_0\ i_1\,)$
$X(\, i_0\ i_2 \mid i_3\ i_1\,)$

The second entry in Table III is obtained from the first by an i-cycle that exchanges the first and second (pivot) digits. The third entry is obtained from the second by an i-cycle that exchanges the second and fourth digits.

For $N = 16$ and $P = 4$ the data exchanges for two sample i-cycles are given in Table IV below.

Table IV : Sample i-cycle communication paths for N=16 and P=4

$X(i_3\ i_2 \mid i_1\ i_0)$ $X(i_3\ i_1 \mid i_2\ i_0)$			$X(i_3\ i_2 \mid i_1\ i_0)$ $X(i_1\ i_2 \mid i_3\ i_0)$	
$i_3 i_2 i_1 i_0$	$i_3 i_1 i_2 i_0$	p	$i_3 i_2 i_1 i_0$	$i_1 i_2 i_3 i_0$
0 0 0 0	0 0 0 0	0	0 0 0 0	0 0 0 0
0 0 0 1	0 0 0 1	0	0 0 0 1	0 0 0 1
0 0 1 0	0 1 0 0	0	0 0 1 0	1 0 0 0
0 0 1 1	0 1 0 1	0	0 0 1 1	1 0 0 1
0 1 0 0	0 0 1 0	1	0 1 0 0	0 1 0 0
0 1 0 1	0 0 1 1	1	0 1 0 1	0 1 0 1
0 1 1 0	0 1 1 0	1	0 1 1 0	1 1 0 0
0 1 1 1	0 1 1 1	1	0 1 1 1	1 1 0 1
1 0 0 0	1 0 0 0	2	1 0 0 0	0 0 1 0
1 0 0 1	1 0 0 1	2	1 0 0 1	0 0 1 1
1 0 1 0	1 1 0 0	2	1 0 1 0	1 0 1 0
1 0 1 1	1 1 0 1	2	1 0 1 1	1 0 1 1
1 1 0 0	1 0 1 0	3	1 1 0 0	0 1 1 0
1 1 0 1	1 0 1 1	3	1 1 0 1	0 1 1 1
1 1 1 0	1 1 1 0	3	1 1 1 0	1 1 1 0
1 1 1 1	1 1 1 1	3	1 1 1 1	1 1 1 1

The i-cycles consist of parallel exchanges of packets with $N/(2P)$ elements. The i-cycle on the left side of Table IV consists of two exchanges. The last two elements in processor 0 are exchanged with the first two elements in processor 1 and the last two elements in processor 2 are exchanged with the first two elements in processor 3. The i-cycle on the right side of Table IV also consists of two exchanges. The last two elements in processor 0 are exchanged with the first two elements in processor 2 and the last two elements in processor 1 are exchanged with the first two elements in processor 3.

The i-cycle has three properties that make it useful for the development of parallel communication algorithms on the hypercube.

I-cycle property A :

An i-cycle may or may not require interprocessor communication, depending on whether or not the digit is in the processor number. For example, the first i-cycle in Table III does not require interprocessor communication because the processor number is unchanged. However the second i-cycle does require interprocessor communication because the processor number is changed. When interprocessor communication is required it is between processors that are directly connected because the processor numbers differ in only one bit. Through this discussion we are assuming that the sequence to processor map is an index-digit permuted map. A direct connection would not be established if the underlying map was (for example) a binary Gray code.

I-cycle property B :

It can be shown that at each stage of the FFT the packets transmitted between processors each contains $2^{r-d-1} = N/(2P)$ elements and that $P/2$ packets are exchanged in parallel.

I-cycle property C :

Any index-digit permutation can be implemented as a sequence of i-cycles. To see this, first decompose the permutation into disjoint cycles. Next decompose each cycle into i-cycles by interchanging the first position with the pivot position and restore it following the completion of the cycle. For example, if the cycle is (2,8,7,5) and the pivot is in position 3, then this cycle is equivalent to the i-cycles (3,2)(3,8)(3,7)(3,5)(3,2) applied from left to right. Any index permutation can be implemented in no more than $1.5d$ i-cycles [7].

2.2 The Standard-order FFT

Consider now the implementation of a standard-order FFT. The i-cycles are given in Table V below for the case $r = 8$ and $d = 5$. The subscripts of the digits are increasing for a transform in standard order like the last entry in Table II. The letter "a" in the superscript indicates an ordering rather than computational step and an "*" following an entry indicates that a parallel transmission was necessary for that step. The sequence of i-cycles is selected based on the theory presented in [7] where it is shown that for even $r > d/2$ a total of $r/2+d+1 = 10$ parallel transmissions are required.

Table V : Intermediate Orderings for a standard order FFT
for $N = 256$ and $P = 32$

$x(i_7\, i_6\, i_5\, i_4\, i_3 \mid i_2\, i_1\, i_0)$
$X^{(1)}(i_2\, i_6\, i_5\, i_4\, i_3 \mid k_7\, i_1\, i_0)^*$
$X^{(2)}(i_2\, k_7\, i_5\, i_4\, i_3 \mid k_6\, i_1\, i_0)^*$
$X^{(3)}(i_2\, k_7\, k_6\, i_4\, i_3 \mid k_5\, i_1\, i_0)^*$
$X^{(4)}(i_2\, k_7\, k_6\, k_5\, i_3 \mid k_4\, i_1\, i_0)^*$
$X^{(5)}(i_2\, k_7\, k_6\, k_5\, k_4 \mid k_3\, i_1\, i_0)^*$
$X^{(5a)}(i_2\, k_7\, k_6\, k_3\, k_4 \mid k_5\, i_1\, i_0)^*$
$X^{(6)}(k_5\, k_7\, k_6\, k_3\, k_4 \mid k_2\, i_1\, i_0)^*$
$X^{(6a)}(k_5\, k_7\, k_2\, k_3\, k_4 \mid k_6\, i_1\, i_0)^*$
$X^{(7)}(k_5\, k_7\, k_2\, k_3\, k_4 \mid k_1\, k_6\, i_0)$
$X^{(7a)}(k_5\, k_1\, k_2\, k_3\, k_4 \mid k_7\, k_6\, i_0)^*$
$X^{(8)}(k_5\, k_1\, k_2\, k_3\, k_4 \mid k_0\, k_6\, k_7)$
$X^{(8a)}(k_0\, k_1\, k_2\, k_3\, k_4 \mid k_5\, k_6\, k_7)^*$

2.3 The A-order FFT

The mapping of a sequence onto the processors is known to significantly influence the time that is required for communication and hence mappings that reduce communication are of considerable interest. The difficulty with selecting a map that minimizes communication for a particular algorithm is that it may not be optimum for a different part of the overall computation. However without knowledge of the other algorithms, and their optimal maps, it is not unreasonable to permit orderings other than the standard order. If order is not a consideration then it is known that the FFT can be performed with d parallel transmissions. However it is likely that the other parts of the overall computation will expect the order of the transform and the input sequence to be the same, particularly if utilities and subroutines are used. Therefore we define an **ordered transform** as any transform in which the order of the sequence and its transform are the same.

In this section we will consider a variant of the parallel FFT presented above in which the input sequence and transform are A-ordered. Communication is reduced and, as mentioned in the introduction, it is just as simple to select this order as the standard order on the Connection Machine using geometry and priorities.

Definition 4 : An **A-order** sequence to processor map $x(i_{d-1} \cdots i_0 \mid i_{r-1} \cdots i_d)$ is one in which the element x_n with $n = i_{r-1}i_{r-2}\cdots i_0$ (binary) has **address** $i_{r-1}i_{r-2} \cdots i_d$ in **processor number** $i_{d-1}i_1 \cdots i_0$.

An A-order FFT is an ordered FFT according to the definition that was given in section 1 and it requires fewer parallel transmissions than a standard-order FFT. An example is given in Table VI below for the case $N = 256$ and $P = 32$. As before, the locations that correspond to the digits on the right of the partition '|' reside in the same physical processor. The digits on the left of the partition correspond to the processor number. An entry that ends with a '*' indicates a parallel transmission and the lines with superscripts that end with a 'a' involve only communication.

Table VI : Intermediate Orderings for an A-order FFT with $N = 256$ and $P = 32$

$x(\, i_4\, i_3\, i_2\, i_1\, i_0 \mid i_7\, i_6\, i_5\,)$
$X^{(1)}(\, i_4\, i_3\, i_2\, i_1\, i_0 \mid k_7\, i_6\, i_5\,)$
$X^{(2)}(\, i_4\, i_3\, i_2\, i_1\, i_0 \mid k_6\, k_7\, i_5\,)$
$X^{(3)}(\, i_4\, i_3\, i_2\, i_1\, i_0 \mid k_5\, k_7\, k_6\,)$
$X^{(4)}(\, k_5\, i_3\, i_2\, i_1\, i_0 \mid k_4\, k_7\, k_6\,)^*$
$X^{(5)}(\, k_5\, k_4\, i_2\, i_1\, i_0 \mid k_3\, k_7\, k_6\,)^*$
$X^{(5a)}(\, k_3\, k_4\, i_2\, i_1\, i_0 \mid k_5\, k_7\, k_6\,)^*$
$X^{(6)}(\, k_3\, k_4\, k_5\, i_1\, i_0 \mid k_2\, k_7\, k_6\,)^*$
$X^{(6a)}(\, k_3\, k_4\, k_5\, i_1\, i_0 \mid k_6\, k_7\, k_2\,)$
$X^{(7)}(\, k_3\, k_4\, k_5\, k_6\, i_0 \mid k_1\, k_7\, k_2\,)^*$
$X^{(7a)}(\, k_3\, k_4\, k_5\, k_6\, i_0 \mid k_7\, k_1\, k_2\,)$
$X^{(8)}(\, k_3\, k_4\, k_5\, k_6\, k_7 \mid k_0\, k_1\, k_2\,)^*$

The communication complexity for an A-order FFT on parallel hypercube is given in the following lemma.

Lemma : An A-order FFT of length $N = 2^r$ can be implemented on a hypercube of dimension d (where $d > r/2$) with $2d - r/2$ parallel transmissions if r is even and $2d - (r-1)/2$ parallel transmissions if r is odd.

Proof :

The normal i-cycles require d parallel transmissions since every physical processor address digit has to be transferred into the pivot position. The extra i-cycles are performed on the most significant $r/2$ digits, $r/2-(r-d)$ of which are located in the processor address. Thus, a total of $d + r/2 - (r-d)) = 2d - r/2$ parallel transmissions is needed. A similar proof can be developed for odd r.

The A-order transform in Table VI requires six parallel transmissions compared with ten for the standard-order FFT in Table V. In general the A-order FFT requires anywhere from d to $1.5d$ parallel transmissions and the standard-order FFT requires anywhere from $1.5d$ to $2d$ parallel transmissions. More specifically, for $d > r/2$, the A-order FFT requires $2d-r/2$ transmissions compared to $d+r/2+1$ for the standard-order FFT. Therefore the A-order FFT requires $r-d+1$ fewer parallel transmissions than the standard-order FFT. For the finest grain computations with $d = r-1$ they differ by only two parallel transmissions. Nevertheless this difference will likely be noticeable because the total communication time is proportional to $O(\log N)$ which is also a small integer.

The FFT is often a part of a larger computation that is posed on a grid so it is reasonable to ask about the compatibility of the Binary Reflected Gray code ordering and A-ordering. In both the standard-order and the A-order transform the processors can be mapped so that nearest neighbors are at a distance of one, but at the expense of the i-cycles being conducted at a distance of two.

2.4 The Algorithm

The parallel hypercube FFT algorithm, written in pseudocode (similar to CM FORTRAN) is included in the following. The variable declaration and initialization have not been included.

```
C  Parallel Hypercube FFT using the A-order Transform
C  K : log2 (N) - 1
     SUBROUTINE FFT
     DO I =  K, 0, -1
       IF (I≠K) CALL ICYCLE (I)                  /* I-cycle */
       CALL CALCULATE_TWIDDLE                    /* Calculate trigonometric factor */
       TEMP = DATA1 + DATA2                      /* Compute new data points */
       DATA2 = (DATA1 - DATA2) * TWIDDLE
       DATA1 = TEMP
       IF (I <= n/2 AND I≠0) THEN                /* Extra I-cycles */
```

```
        CALL ICYCLE (n-I-1)
      END IF
    END DO
    END
```

3. Computing the Trigonometric Coefficients

There are a few alternative methods for computing the trigonometric coefficients depending on the available memory, I/O bandwidth, and processing capabilities [3].

a. **Recursion** All of the trigonometric coefficients at each stage are generated by recursion. This scheme requires only $O(1)$ storage and is popular on a uniprocessor or vector processors. However, the computation is highly sequential and not suitable for multiprocessors.

b. **Table look-up** The trigonometric coefficients are precomputed and stored in each processor. This scheme has an advantage for many FFTs since the trigonometric coefficients would be available for use without recalculation. However, this scheme also requires a large amount of memory proportional to $\log N$ in each of the N processors. This may not be desirable for massive parallel computers where memory is limited.

c. **Direct calculation** The trigonometric coefficients can be computed directly from the equation $W^{-k} = cos(2k\pi/N) - i\ sin(2k\pi k/N)$. However, the calculation of the trigonometric functions on each stage is very time consuming. Particularly since the FFT itself requires only a few operations.

d. **Permutation** Initially, the trigonometric coefficients are distributed among the processors according to the calculations required in the first stage. In the subsequent stages, half of the trigonometric coefficients are permuted each to two other processors. This scheme may be inefficient on parallel machine such as the Connection Machine where communication is expensive.

None of these methods are completely satisfactory on massively parallel computers if memory is limited and communication is expensive. However, by performing a few additional operations at each stage, the trigonometric coefficients can be computed in parallel without any communication.

Consider the following example of a 16-point FFT (unordered transform) and suppose that element i is mapped to processor i, then the trigonometric factors needed at each stage are as in Table VII below. The entries in each column correspond to k in the trigonometric factor W^{-k}. Entries with the form (k) refer to the exponent of a coefficient that is not used at the current stage but is needed to compute the coefficients at a subsequent stage of the FFT.

Table VII : Trigonometric Coefficients for a 16-point unordered FFT

Processor	Value of k in W^{-k}			
Processor Number (binary)	Stage 1	Stage 2	Stage 3	Stage 4
0000	(0)	(0)	(0)	(0)
0001	(1)	(2)	(4)	0
0010	(2)	(4)	0	(0)
0011	(3)	(6)	4	0
0100	(4)	0	(0)	(0)
0101	(5)	2	(4)	0
0110	(6)	4	0	(0)
0111	(7)	6	4	0
1000	0	(0)	(0)	(0)
1001	1	(2)	(4)	0
1010	2	(4)	0	(0)
1011	3	(6)	4	0
1100	4	0	(0)	(0)
1101	5	2	(4)	0
1110	6	4	0	(0)
1111	7	6	4	0

It can be seen that the integers in each column are twice (mod N/2) the integers in the previous column and hence the trigonometric coefficients can be computed from the identities.

$$\cos 2\theta = \cos^2\theta - \sin^2\theta \quad \text{, and} \tag{3}$$

$$\sin 2\theta = 2 \cos \theta \sin \theta \, . \qquad (4)$$

Thus, we can calculate the trigonometric coefficients for the current stage from the previous stage by four multiplications and one addition (or three multiplications and two additions). This method can also be used to generate the table for the table look-up scheme. It can also be used to compute the coefficients for the ordered (both A-order and standard-order) parallel hypercube FFT presented in section 2 with a slight modification for the initial trigonometric factor calculations. Table VIII below contains the exponents for the A-order transform with N=16. An initial standard sequence to processor map is assumed.

Table VIII : Trigonometric Coefficients for a 16-point parallel hypercube FFT using A-order and i-cycles

Processor	Value of k in W^{-k}			
Processor Number (binary)	Stage 1	Stage 2	Stage 3	Stage 4
0000	-	-	-	-
0001	-	-	-	-
0010	-	-	-	-
0011	-	-	-	-
0100	-	-	-	-
0101	-	-	-	-
0110	-	-	-	-
0111	-	-	-	-
1000	0	0	0	0
1001	1	2	4	0
1010	2	4	0	0
1011	3	6	4	0
1100	4	0	0	0
1101	5	2	4	0
1110	6	4	0	0
1111	7	6	4	0

Fewer computations are required because every trigonometric coefficient is used and therefore a factor of two is saved compared to the unordered FFT. In general, this method of computing trigonometric coefficients can be used if the order of the not-yet-transformed bits (i_j) is preserved. The characteristics of the methods for

computing the trigonometric coefficients are summarized in Table IX below.

Table IX : Characteristics of Different Methods for Computing Trigonometric Coefficients

Method	storage	computation	communication	comment
recursion	$O(1)$	$O(N \log N)$	0	highly sequential
table look up	$O(N \log N)$	$O(\log N)$	0	reuseability
permutation	$O(N)$	$O(1)$	$O(\log N)$	--
direct calculation	$O(N)$	$O(\log N)$	0	use sin & cos
new method	$O(N)$	$O(\log N)$	0	no sin & cos

4. Performance of the Parallel Hypercube FFTs on the CM-2

4.1 Performance results for the TMC FFT

Consider first the performance of the TMC FFT that is currently available on the Connection Machine. The execution times of both the ordered and unordered FFT is presented in table X. FFT (A) and FFT (B) correspond to the unordered and ordered FFTs respectively and the results were obtained on a 32k processor CM-2. The entry '--' means that the result could not be computed because it required more memory than what was available. The MFLOPS are computed from the formula MFLOPS = $5N\log N$/time which does not include the precomputed trigonometric coefficients.

Table X : Execution times for TMC FFT (32k)

size FFT	FFT (A) (sec)	MFLOPS(32k)	FFT (B) (sec)	MFLOPS(32k)
65536	0.02	262	0.03	175
131072	0.04	279	0.08	139
262144	0.09	262	0.22	107
524288	0.17	293	0.56	89
1048576	0.35	300	1.79	59
2097152	0.69	319	6.21	35
4194304	1.40	330	--	--
8388608	2.81	343	--	--

FFT (A) is the TMC FFT without bit-reversal

FFT (B) is the TMC FFT with bit-reversal

-- memory was exceeded

The difference between the time for FFT (A) and FFT (B) is due to the additional communication that is required to bit-reverse the results of FFT(A). From the table it is clear that performing bit-reversal is expensive and that performance deteriorates for larger problems.

4.2 Performance of a CM FORTRAN version of the standard-order FFT

In this subsection we will examine the performance of the standard-order FFT using i-cycles in the intermediate phases of the algorithm. The program was written in the beta release version of the CM FORTRAN with partial optimization using compiler options. At present, the system software will use a binary reflected Gray code mapping of the logical processors onto the physical processors. Therefore most i-

cycles will communicate over a physical distance (Hamming distance) of two which requires twice the communication of a map in which the logical and physical processors have the same number. The latter case will be discussed in the next subsection.

The execution times and MFLOPS for the FORTRAN version are listed in Table XI.

Table XI : Execution times for the CM FORTRAN standard-order FFT (32k)

size FFT	Execution time (sec)	MFLOPS(32k)
65536	0.08	98
131072	0.16	104
262144	0.32	111
524288	0.66	113
1048576	1.34	117
2097152	2.81	118
4194304	5.67	122
8388608	11.68	124

The MFLOPS in Table XI above are calculated from MFLOPS = $7.5N\log N$/time (which includes 2.5 N log N operations for computing the trigonometric coefficients). Comparing Table X and XI it can be observed that for small N, the ordered TMC FFT is about twice as fast as the standard-order FFT, (e.g. 0.08 sec versus 0.16 sec for 131072-point FFT). However for large N, the standard-order FFT using i-cycles outperforms the ordered TMC FFT (e.g. 2.81 sec versus 6.21 sec for 2M-point FFT). Also, from Table X, the execution times for FFT (B) triples when the size of the input doubles. On the other hand, from Table XI, the execution times for standard-order

FFT using i-cycles approximately doubles when the size of the input doubles.

From these comparisons we conclude that the standard-order FFT using i-cycles provides enhanced performance compared to an FFT with separate bit-reversal and butterfly phases. It should be mentioned that the TMC FFT was written in lower level languages while the results in Table XI were obtained with a high level language (CM FORTRAN) which is also in its beta release. Thus, further improvement is expected for an implementation in a optimized low level languages or with a mature FORTRAN compiler.

Even though the FFT has been implemented with an efficient communication algorithm using i-cycles, over 80 percent of the execution time is still spent in communication. In the next section, communication will be further reduced by avoiding the binary reflected Gray code mapping of the logical to physical processors.

4.3 A Comparison of three FFTs on the Connection Machine.

In the previous subsection we examined the performance of a CM FORTRAN version of the FFT in which the binary reflected Gray code was used to map logical processors to physical processors. Although this map is ideal for nearest neighbor communication, it slows the i-cycle communication for the FFT by a factor of two. In this section we will consider the performance of three ordered FFTs on a hypercube whose logical and physical processor numbers are the same.

1. The standard-order FFT which combines the bit-reversal and the butterfly phases.
2. The A-order FFT which also combines the bit-reversal and the butterfly phases.
3. An FFT written by Hertz [2] which separates the bit-reversal and the butterfly phases.

Using CM FORTRAN/PARIS it is possible to equate logical and physical processor numbers. That is, any reference to processor $i_{d-1} \cdots i_0$ is a reference to a processor with the same binary representation in the hypercube and not to a processor whose

number is the binary reflected Grey code map of $i_{d-1} \cdots i_0$. A significant improvement is obtained because the key communication task (i-cycle) is conducted at a physical distance of at most one using *news* communication for all i-cycles. The programs were written in CM FORTRAN/PARIS and run on a 32k CM-2. The times for different size FFT are listed in Table XII and the corresponding MFLOPS counts are listed in Table XIII.

Table XII : Computing time in seconds for three ordered FFTs

size FFT	machine size	FFT (1)	FFT (2)	FFT (3)
131072	8k	0.22	0.16	--
262144	8k	0.45	0.32	--
524288	8k	0.94	0.67	--
1048576	8k	1.92	1.39	--
2097152	8k	3.95	2.89	--
262144	16k	0.23	0.17	0.688
524288	16k	0.49	0.36	1.40
1048576	16k	1.01	0.72	2.95
2097152	16k	2.07	1.50	6.10
4194304	16k	4.23	3.07	12.68
524288	32k	0.25	0.19	--
1048576	32k	0.52	0.39	--
2097152	32k	1.09	0.80	--
4194304	32k	2.22	1.59	--
8388608	32k	4.55	3.29	--

FFT (1) standard order FFT.

FFT (2) A-order FFT.

FFT (3) P. Hertz FFT [2].

Table XIII : MFLOPS for three ordered FFTs

size FFT	machine size	FFT (1)	FFT (2)	FFT (3)
131072	8k	76	104	--
262144	8k	79	111	--
524288	8k	79	112	--
1048576	8k	82	113	--
2097152	8k	84	114	--
262144	16k	154	208	51
524288	16k	152	208	53
1048576	16k	156	218	53
2097152	16k	160	220	54
4194304	16k	164	225	55
524288	32k	299	393	--
1048576	32k	302	403	--
2097152	32k	303	413	--
4194304	32k	318	435	--
8388608	32k	318	440	--

FFT (1) standard order FFT.

FFT (2) A-order FFT.

FFT (3) P. Hertz FFT [2].

-- data not available

Note : The MFLOPS for (3) is calculated using the same formula as (1) and (2). In reality, method (3) requires more than 7.5 operation per point and thus the MFLOPS count should be higher.

These results demonstrate the attributes of A-ordering, i-cycles, and the new parallel method of computing the trigonometric coefficients. From Table XIII, we estimate a performance of about .9 GFLOPS for a 16M-point FFT on a full 64k CM-2.

5. Summary and Conclusion

First, the experimental results in section 4 demonstrate that performance can be improved by using the ordered parallel FFTs that reduce communication by combining the communication and computational phases [7]. Although this result has been demonstrated on the Connection Machine it would also be true for any hypercube because communication time is a significant part of the overall computing time. Second, the A-order FFT has performance that is superior to the standard-order FFT and is therefore recommended where applicable. In addition, a parallel algorithm for computing the trigonometric coefficients was presented that represents an attractive compromise between the communication, computation, and memory constraints that exist on the Connection Machine. The use of the i-cycle, A-ordering, and the new parallel algorithm for computing the trigonometric coefficients have resulted in the development of a high performance ordered FFT for the Connection Machine.

References :

1. D. Fraser, "Array permutation by index-digit permutation", *J. ACM*, 22(1976), pp. 298-306.

2. P. Hertz, "An Algorithm for the Fast Fourier Transform On the Connection Machine", accepted by *Computers in Physics*, June 1989.

3. R.A. Kamin III, and G.B. Adams III, "Fast Fourier Transform Algorithm Design and Tradeoffs on the CM", *Proceedings of the Conference on Scientific Applications of the Connection Machine*, Editor : H. Simon, World Scientific Publishing Co., 1989.

4. O.A. McBryan, "Connection Machine Application Performance", CU-CS-434-89, Department of Computer Science, University of Colorado, April 1989.

5. A.V. Oppenheim, and R.W. Schafer, *Digital Signal Processing*, Prentice Hall, 1975.

6. P.N. Swarztrauber, "FFT algorithms for vector computers", *Parallel Computing*, 1 (1984), pp. 45-63.

7. P.N. Swarztrauber, "Multiprocessor FFTs", *Parallel Computing*, 5 (1987), pp. 197-210.

Biographies

Charles H. Tong is a graduate student in the Computer Science Department at the University of California, Los Angeles. He received his B.S. degree in Electrical Engineering and Computer Science at U.C. Berkeley in 1982 and M.S. degree in Electrical and Computer Engineering at U.C. Davis in 1986. He worked as a test system engineer in Intel Corporation from August 1982 to August 1985. His research interests are parallel numerical solution of partial differential equations, parallel algorithms for numerical linear algebra, parallel computer architectures, and systolic arrays for numerical solutions of sparse linear systems.

Paul N. Swarztrauber is a Senior Scientist at the National Center for Atmospheric Research and a Adjoint Professor in the Computer Science Department at the University of Colorado. His research interests are in computational mathematics including the numerical solution of partial differential equations, parallel algorithms for numerical linear algebra, harmonic analysis, parallel and vector algorithms for the fast Fourier transform and numerical software.